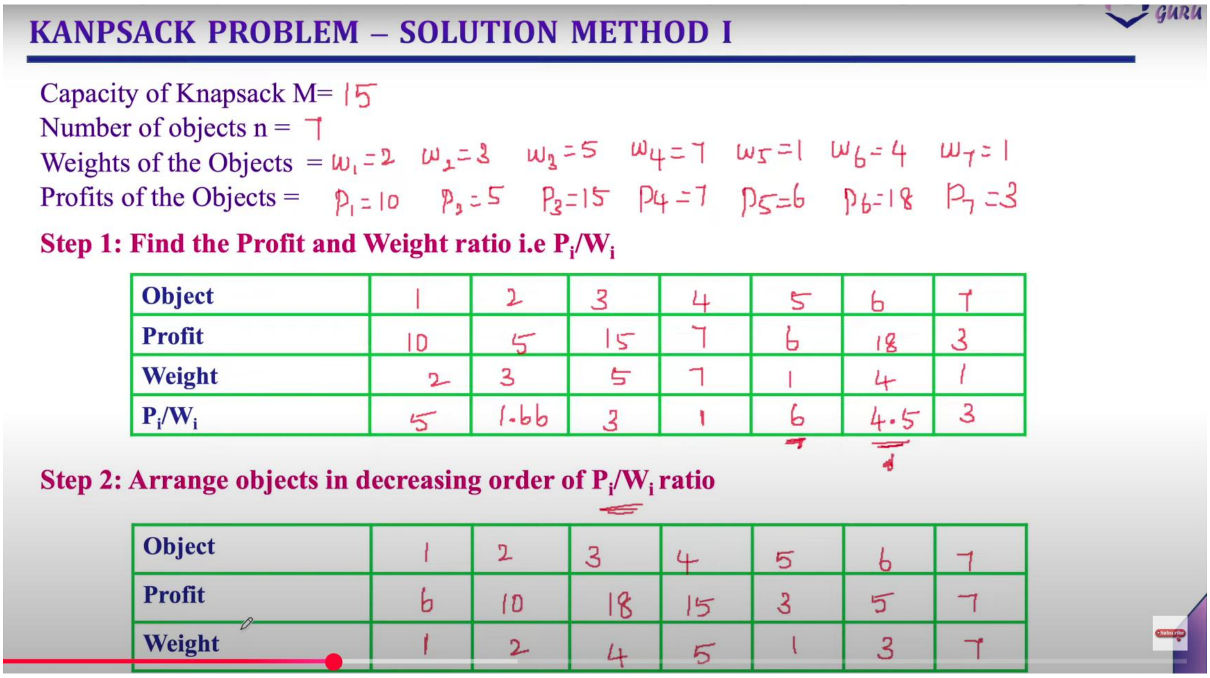
Unit 3 Greedy Method

Fractional Knapsack



KANPSACK PROBLEM – SOLUTION METHOD I

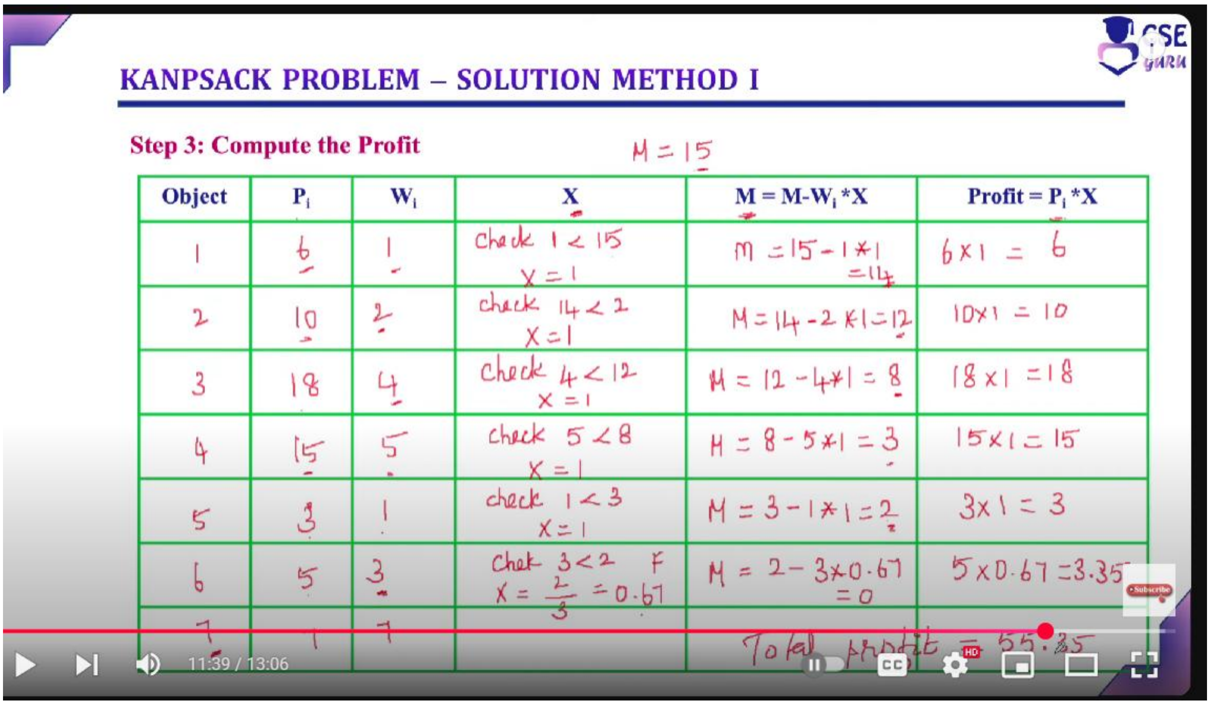
Capacity of Knapsack M= 15
Number of objects n = 7
Weights of the Objects = $w_1=2$ $w_2=3$ $w_3=5$ $w_4=7$ $w_5=1$ $w_6=4$ $w_7=1$
Profits of the Objects = $p_1=10$ $p_2=5$ $p_3=15$ $p_4=7$ $p_5=6$ $p_6=18$ $p_7=3$

Step 1: Find the Profit and Weight ratio i.e P_i/W_i

Object	1	2	3	4	5	6	7
Profit	10	5	15	7	6	18	3
Weight	2	3	5	7	1	4	1
P_i/W_i	5	1.66	3	1	6	4.5	3

Step 2: Arrange objects in decreasing order of P_i/W_i ratio

Object	1	2	3	4	5	6	7
Profit	6	10	18	15	3	5	7
Weight	1	2	4	5	1	3	7



KANPSACK PROBLEM – SOLUTION METHOD I

Step 3: Compute the Profit M = 15

Object	P_i	W_i	X	$M = M-W_i*X$	$Profit = P_i*X$
1	6	1	Check 1 < 15, X = 1	M = 15 − 1*1 = 14	6x1 = 6
2	10	2	check 14 < 2, X = 1	M = 14 − 2 x 1 = 12	10x1 = 10
3	18	4	Check 4 < 12, X = 1	M = 12 − 4*1 = 8	18x1 = 18
4	15	5	check 5 < 8, X = 1	M = 8 − 5*1 = 3	15x1 = 15
5	3	1	check 1 < 3, X = 1	M = 3 − 1*1 = 2	3x1 = 3
6	5	3	Chek 3 < 2 F, X = 2/3 = 0.67	M = 2 − 3x0.67 = 0	5x0.67 = 3.35
7	[illegible]	7			

Total profit = 55.35

11:39 / 13:06

Youtube link :- https://youtu.be/_vHCtwx-EW4?feature=shared plaintext

```
function fractionalKnapsack(W, items[1...n]):
    // Each item has weight[i] and value[i]
    // Step 1: Calculate value/weight ratio for each item
    for i from 1 to n:
        ratio[i] = value[i] / weight[i]

    // Step 2: Sort items in descending order of ratio
    sort items by ratio[i] in descending order
```

```
    totalValue = 0
    for i from 1 to n:
        if W == 0:
            break
        if weight[i] <= W:
            W = W - weight[i]
            totalValue = totalValue + value[i]
        else:
            fraction = W / weight[i]
            totalValue = totalValue + (value[i] * fraction)
            W = 0

    return totalValue
```

✅ Time Complexity of Fractional Knapsack (Greedy Approach)

Let:

- `n` = number of items

Steps & Time Complexity:

1. **Compute value/weight ratio for all items:**
   - `O(n)` (one pass through the items)
2. **Sort items by ratio in descending order:**
   - `O(n log n)` (typical sorting)
3. **Iterate over sorted items to fill the knapsack:**
   - `O(n)` (one more pass)

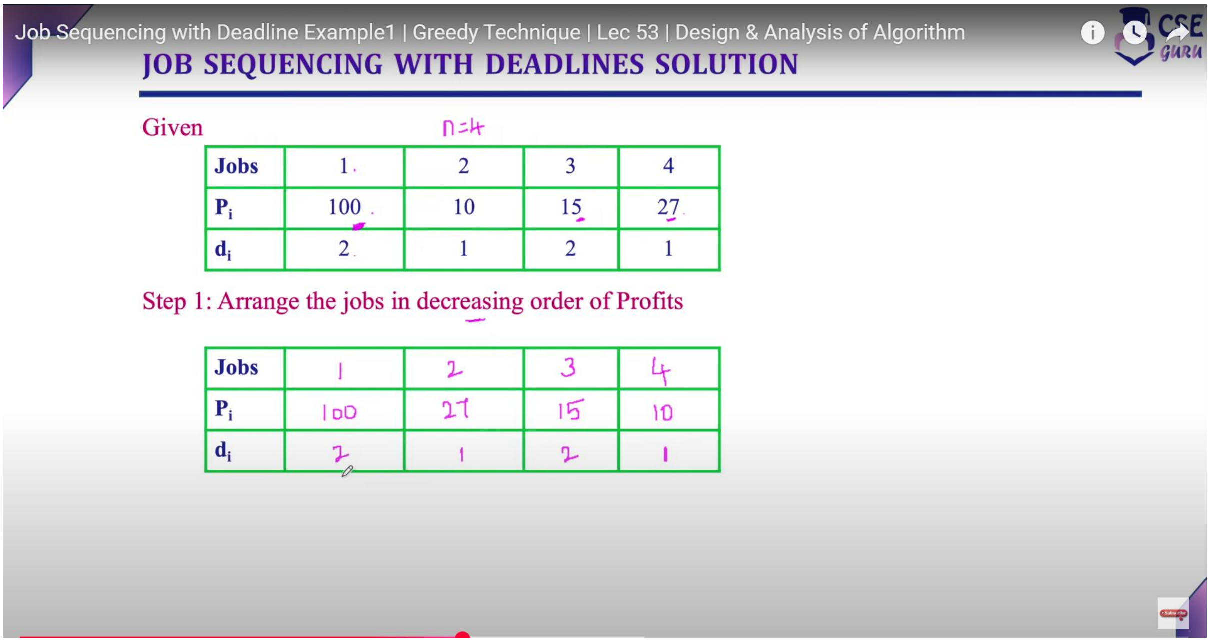
Total Time Complexity:

```plaintext
O(n + n log n + n) = O(n log n)
```

✅ Final Answer:

> Time Complexity = O(n log n)

Job sequencing with deadline



Job Sequencing with Deadline Example1 | Greedy Technique | Lec 53 | Design & Analysis of Algorithm

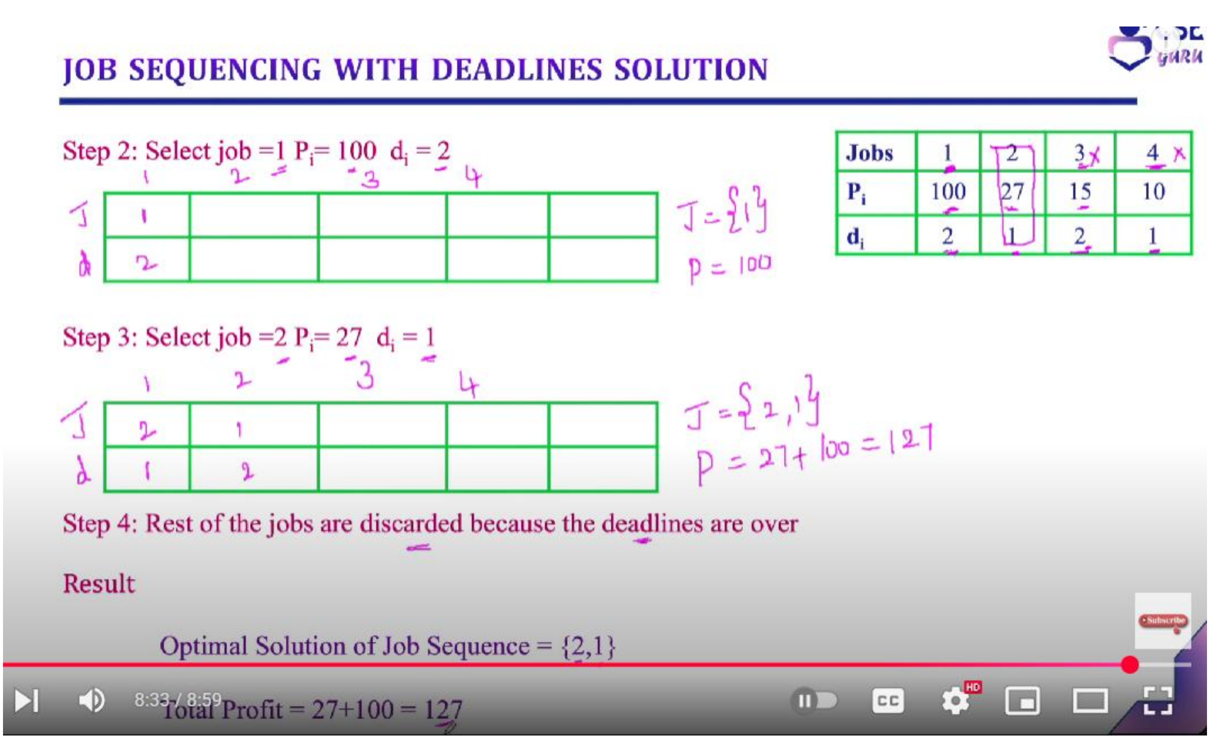
JOB SEQUENCING WITH DEADLINES SOLUTION

Given $n=4$

Jobs	1	2	3	4
P_i	100	10	15	27
d_i	2	1	2	1

Step 1: Arrange the jobs in decreasing order of Profits

Jobs	1	2	3	4
P_i	100	27	15	10
d_i	2	1	2	1



JOB SEQUENCING WITH DEADLINES SOLUTION

Jobs	1	2	3	4
P_i	100	27	15	10
d_i	2	1	2	1

Step 2: Select job =1 P_i= 100 d_i = 2

J	1				
d	2				

$J = \{1\}$
$P = 100$

Step 3: Select job =2 P_i= 27 d_i = 1

J	2	1			
d	1	2			

$J = \{2, 1\}$
$P = 27 + 100 = 127$

Step 4: Rest of the jobs are discarded because the deadlines are over

Result

Optimal Solution of Job Sequence = {2,1}

Total Profit = 27+100 = 127

8:33 / 8:59

JOB SEQUENCING WITH DEADLINES SOLUTION

Given

Job	1	2	3	4	5
P_i	20	15	10	5	1
d_i	2	2	1	3	3

Step 1: Arrange the jobs in decreasing order of Profits

Job	1	2	3	4	5
P_i	20	15	10	5	1
d_i	2	2	1	3	3

JOB SEQUENCING WITH DEADLINES SOLUTION

Job	1	2	3	4	5
P_i	20	15	10	5	1
d_i	2	2	1	3	3

Step 2: Select job =1 P_i= 20 d_i = 2

	1	2	3	4	5	6
J	1					
d	2					

J = {1}

Profit = 20

Step 3: Select job =2 P_i= 15 d_i = 2

	1	2	3	4	5	6
J	1	2				
d	2	2				

J = {1, 2}

Profit = 20 + 15 = 35

Step 4: Select job =3 P_i= 10 d_i = 1. This job is not selected deadline already over

	1	2	3	4	5	
J	1	2				
d	2	2				

J = {1, 2}

JOB SEQUENCING WITH DEADLINES SOLUTION

Job	1	2	~~3~~	4	~~5~~
P_i	20	15	10	5	1
d_i	2	2	1	3	3

Step 5: Select job =4 P_i= 5 d_i = 3

	1	2	3	4	5	
J	1	2	4			
d	2	2	3			

J = {1, 2, 4}

profit = 20 + 15 + 5 = 40

Step 6: Rest of the jobs are discarded because the deadlines are over

Result

Optimal Solution of Job Sequence = {1, 2, 4}

Total Profit = 40

Youtube links :- https://youtu.be/rXg3l8__iUE?feature=shared

https://youtu.be/5uIQ5Z4u-vE?feature=shared

📌 Job Sequencing with Deadline – Pseudocode

Assumption:

Each job has:

- `id[i]` = job identifier
- `deadline[i]` = last day by which it must be done
- `profit[i]` = profit earned if job is completed before or on its deadline

plaintext Copy Edit

```plaintext
function jobSequencing(jobs[1...n]):
    // Each job has: id, deadline, profit

    // Step 1: Sort all jobs in decreasing order of profit
    sort jobs by profit in descending order

    // Step 2: Find maximum deadline to create time slots
    maxDeadline = maximum of all job.deadline

    // Step 3: Create a slot array to mark time slots
    slots[1...maxDeadline] = all false  // initially all slots are free

    // Step 4: For each job, find a free slot before its deadline
    totalProfit = 0
    resultJobs = []
```

```plaintext
    for job in jobs:
        for j = job.deadline down to 1:
            if slots[j] == false:
                slots[j] = true  // mark this slot as filled
                resultJobs.append(job.id)
                totalProfit = totalProfit + job.profit
                break

    return resultJobs, totalProfit
```

✅ Time Complexity of Job Sequencing with Deadlines (Greedy Approach)

Let:

- `n` = number of jobs
- `d` = maximum deadline among all jobs

Steps & Time Complexity:

1. **Sort jobs by profit in descending order:**
   - `O(n log n)` (standard comparison-based sort)

2. **Iterate over each job and find a free time slot ≤ deadline:**
   - **Naive method (check from deadline down to 1):**
     - Worst case: scan up to `d` slots for each job
     - `O(n × d)` → if `d = O(n)` ⇒ $O(n^2)$
   - **Optimized method using Disjoint Set (Union-Find):**
     - Finding next free slot: `O(log n)` using path compression
     - Total time: `O(n log n)`

🔚 Total Time Complexity:

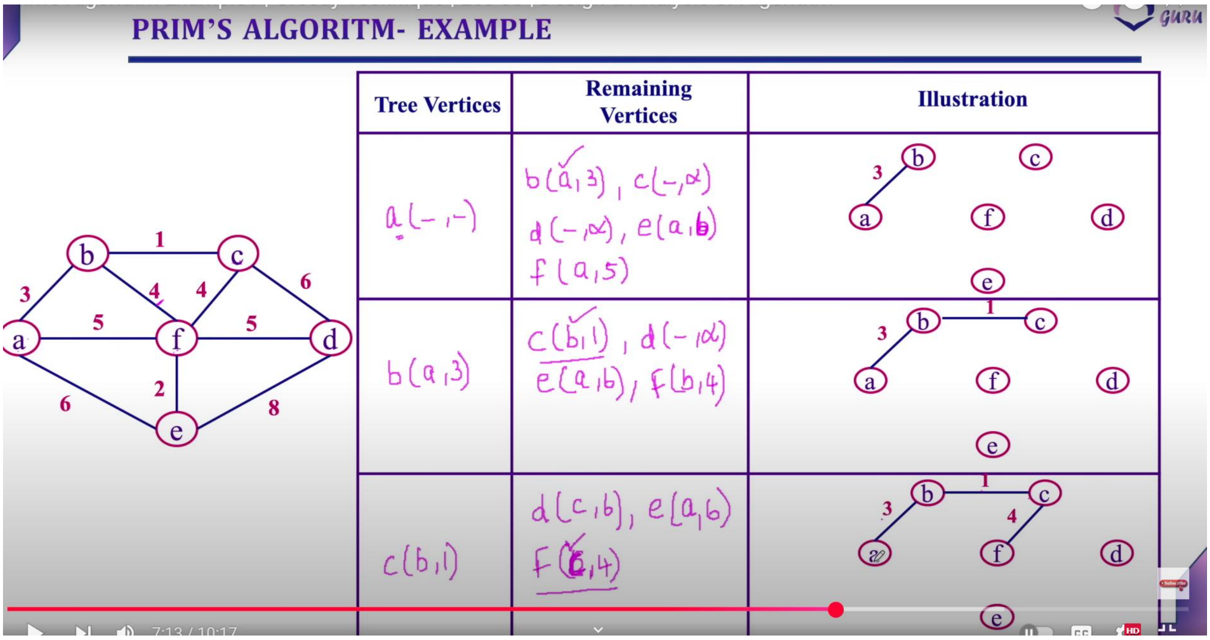
🔸 **Naive Version:**

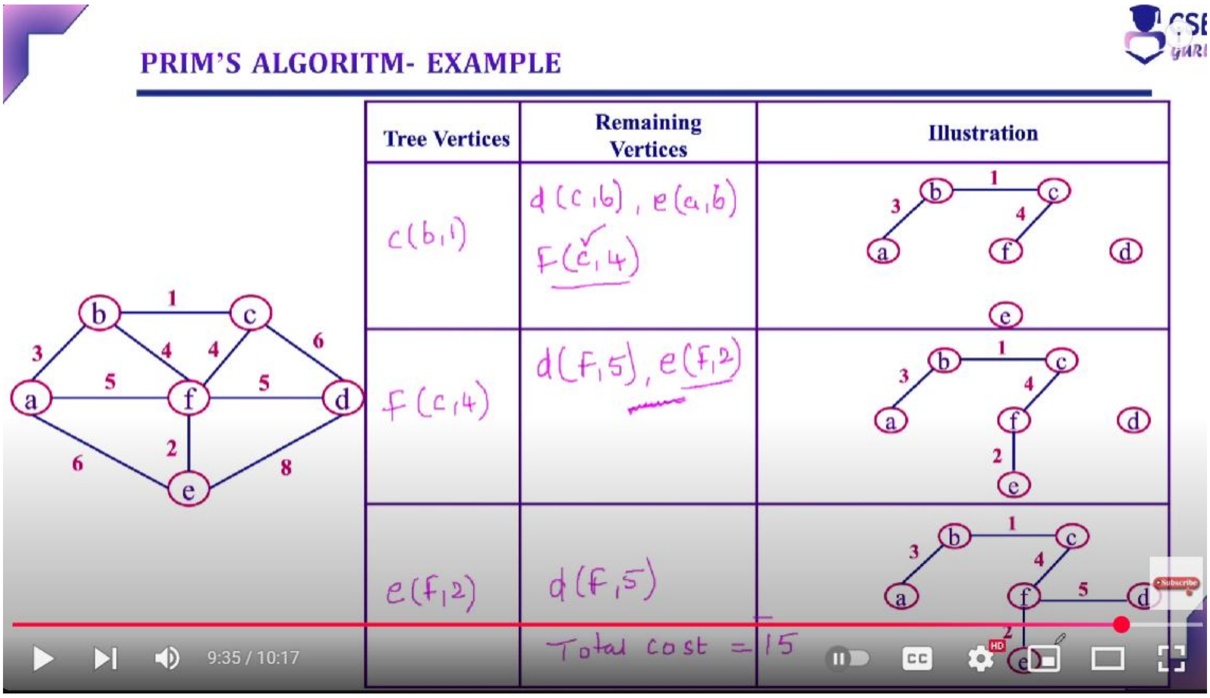
```plaintext
O(n log n + n × d) → O(n²) when d = O(n)
```

🔹 **Optimized Version (with DSU):**

```plaintext
O(n log n + n log n) = O(n log n)
```

Prim’s Algorithm





Youtube link :- https://youtu.be/uOpTIXrMv1I?feature=shared plaintext

```
function primMST(graph):
    n = number of vertices in graph
    visited[0...n-1] = all false
    minEdgeCost[0...n-1] = ∞
    parent[0...n-1] = -1

    minEdgeCost[0] = 0  // Start from node 0
    priorityQueue = empty min-heap
    push (0, 0) into priorityQueue  // (cost, vertex)

    while priorityQueue is not empty:
        (cost, u) = extract_min(priorityQueue)

        if visited[u] == true:
            continue
        visited[u] = true

        for each (v, weight) in graph[u]:  // Adjacent vertices
            if visited[v] == false and weight < minEdgeCost[v]:
                minEdgeCost[v] = weight
                parent[v] = u
                push (weight, v) into priorityQueue

    return parent  // or use parent[] to construct MST edges
```

✅ Time Complexity of Prim's Algorithm

Let:

- `V` = number of vertices
- `E` = number of edges

Two Common Implementations & Time Complexities:

🔸 **1. Using Adjacency Matrix + Linear Search** (Naive)

1. **Pick minimum weight unvisited vertex:**
   - `O(V)` for each of `V` vertices → `O(V²)`
2. **Update adjacent vertices:**
   - Each update in constant time → `O(V)` per v ex

🔚 Total Time (Naive):

plaintext Copy Edit

```
O(V²)
```

🔹 **2. Using Adjacency List + Min-Heap (Priority Queue)** (Optimized)

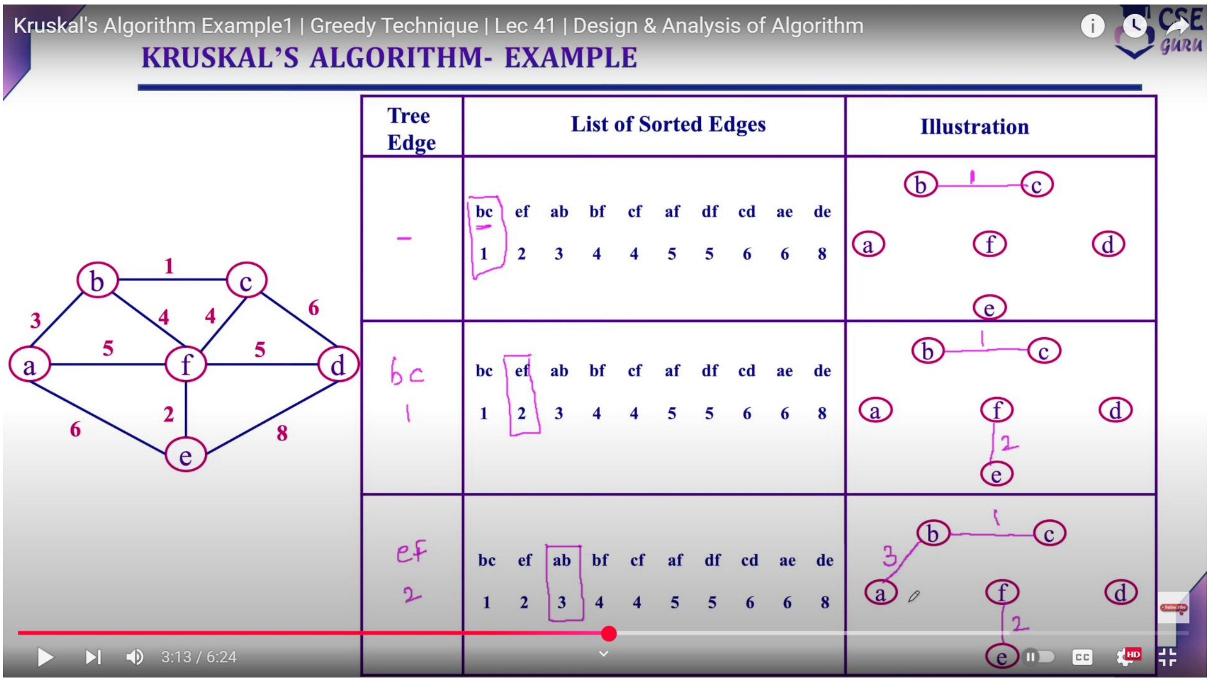
1. **Priority Queue operations** (insert/update min):
   - `O(log V)` per edge
2. **Each edge is processed once:**
   - `O(E log V)`

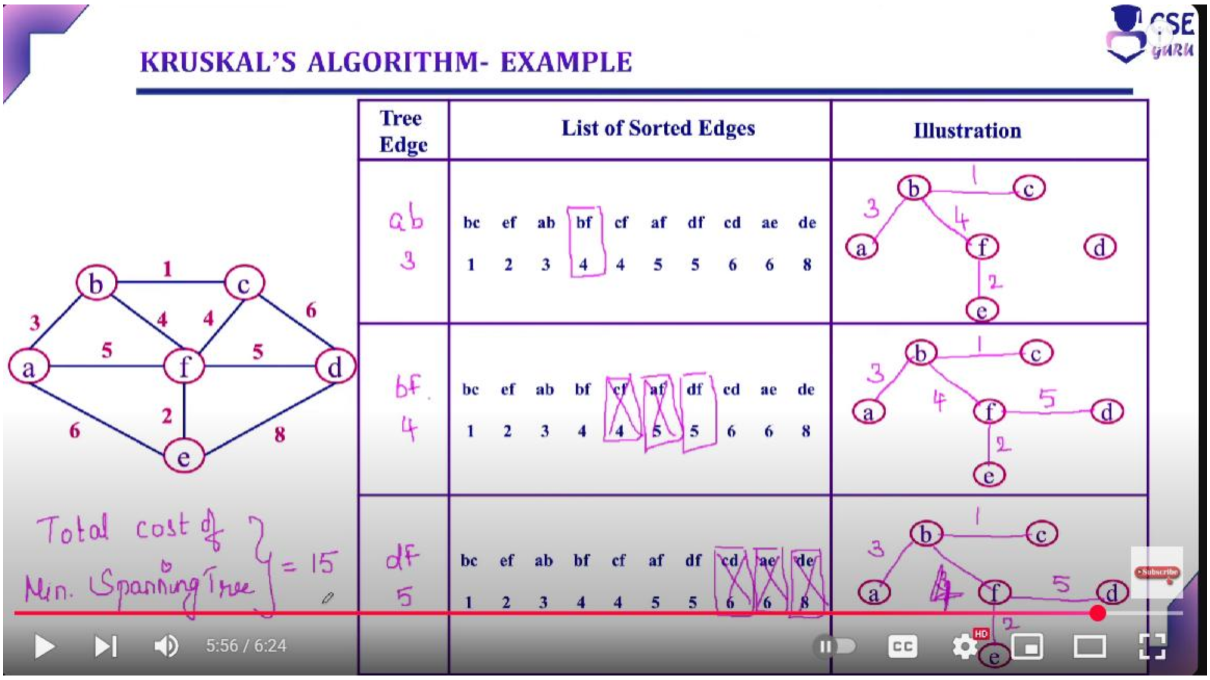
🔚 Total Time (Heap-based):

plaintext Copy Edit

```
O(E log V)
```

Kruskal's algorithm





Youtube link :- https://youtu.be/l3xnyVjKvLg?feature=shared

📌 Kruskal's Algorithm – Pseudocode

plaintext

Copy Edit

```
function kruskalMST(graph):
    n = number of vertices
    edges = list of all edges in the form (u, v, weight)

    // Step 1: Sort all edges by increasing weight
    sort(edges by weight ascending)

    // Step 2: Initialize Disjoint Set Union (DSU)
    for i from 0 to n - 1:
        parent[i] = i
        rank[i] = 0

    mst = []          // List to store MST edges
    totalWeight = 0   // Total weight of MST
```

Copy Edit

```
    // Step 3: Pick edges in increasing weight order
    for each (u, v, weight) in edges:
        if findParent(u) != findParent(v):
            mst.append((u, v, weight))
            totalWeight += weight
            union(u, v)

    return mst, totalWeight


function findParent(x):
    if parent[x] != x:
        parent[x] = findParent(parent[x])  // Path compression
    return parent[x]
```

Copy Edit

```
function union(x, y):
    px = findParent(x)
    py = findParent(y)
    if rank[px] < rank[py]:
        parent[px] = py
    else if rank[px] > rank[py]:
        parent[py] = px
    else:
        parent[py] = px
        rank[px] += 1
```

✅ Time Complexity of Kruskal's Algorithm

Let:

- `V` = number of vertices
- `E` = number of edges

Steps & Time Complexity:

1. **Sort all edges by weight:**
   - Sorting `E` edges → `O(E log E)`
2. **Initialize Disjoint Set (Union-Find):**
   - `O(V)` (to initialize `parent[]` and `rank[]` arrays)
3. **Process each edge and apply Union-Find:**
   - Each edge → `find` and possibly `union`
   - With **path compression + union by rank**, each operation is **O(α(V))** (where α is the inverse Ackermann function, which is nearly constant)
   - Total = `O(E × α(V)) ≈ O(E)`

🔚 Total Time Complexity:

plaintext

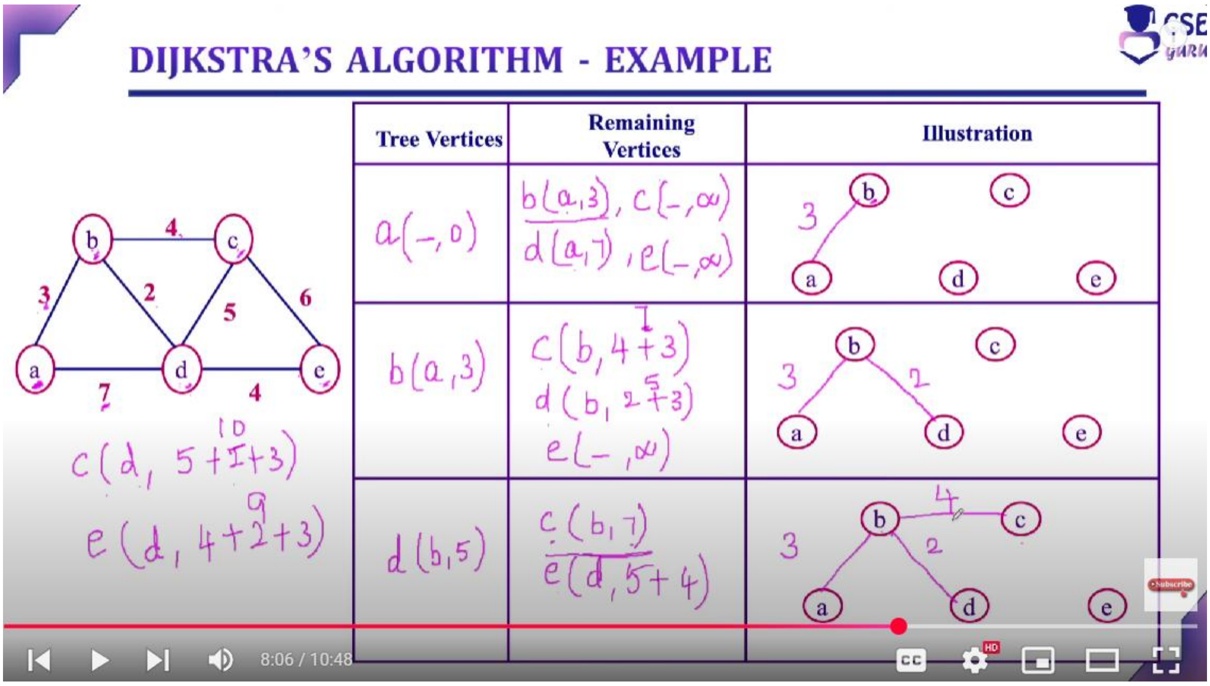
```
O(E log E + E × α(V)) ≈ O(E log E)
```

> Since α(V) is extremely small (≤ 4 for all practical cases), we treat it as constant.

✅ Final Answer:

> **Time Complexity = O(E log E)**

Djiikstra's algorithm

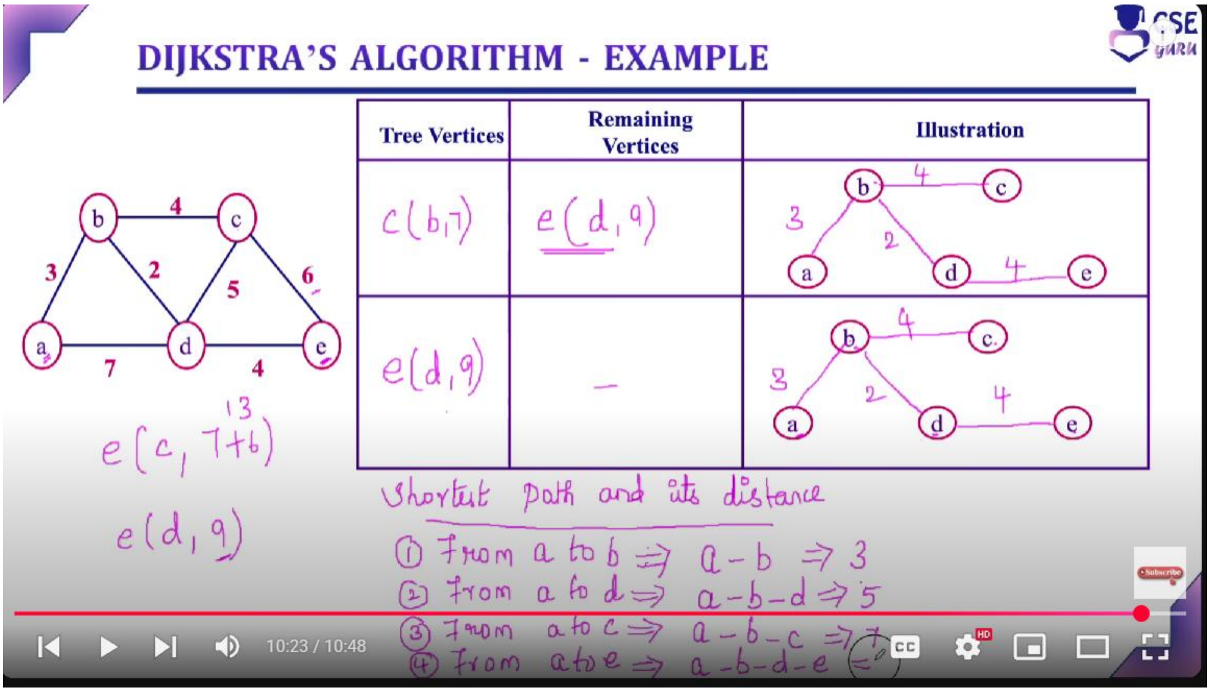


DIJKSTRA'S ALGORITHM - EXAMPLE

Tree Vertices	Remaining Vertices	Illustration
a(−,0)	b(a,3), c(−,∞) d(a,7), e(−,∞)	
b(a,3)	c(b,4+3) d(b,2+3) e(−,∞)	
d(b,5)	c(b,7) e(d,5+4)	

$\overset{10}{c(d, 5+I+3)}$

$\overset{9}{e(d, 4+2+3)}$



DIJKSTRA'S ALGORITHM - EXAMPLE

Tree Vertices	Remaining Vertices	Illustration
c(b,7)	e(d,9)	
e(d,9)	–	

$\overset{13}{e(c, 7+6)}$

e(d,9)

Shortest path and its distance

① From a to b ⇒ a − b ⇒ 3
② From a to d ⇒ a − b − d ⇒ 5
③ From a to c ⇒ a − b − c ⇒ 7
④ From a to e ⇒ a − b − d − e ⇒

📌 Dijkstra's Algorithm – Pseudocode (Using Min-Heap / Priority Queue)

plaintext

```
function dijkstra(graph, source):
    n = number of vertices in graph
    dist[0...n-1] = ∞         // Distance from source to each node
    visited[0...n-1] = false
    dist[source] = 0

    // Min-heap priority queue: (distance, node)
    priorityQueue = empty min-heap
    push (0, source) into priorityQueue
```

```
    while priorityQueue is not empty:
        (currDist, u) = extract_min(priorityQueue)

        if visited[u]:
            continue

        visited[u] = true

        for each (v, weight) in graph[u]:  // Explore neighbors
            if not visited[v] and dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
                push (dist[v], v) into priorityQueue

    return dist
```

✅ Time Complexity of Dijkstra's Algorithm

Let:

- `V` = number of vertices
- `E` = number of edges

Steps & Time Complexity:

🔸 **1. Initialization:**

- Distance array and visited array → `O(V)`

🔸 **2. Extract min node from priority queue:**

- Each vertex is extracted **once** → `O(V log V)`

🔸 **3. For each edge, perform relaxation:**

- For each of the `E` edges, we may update the distance and push into the priority queue → `O(E log V)`

🔚 Total Time Complexity:

```plaintext
O(V log V + E log V) = O((V + E) log V)
```

✅ Final Answer:

> **Time Complexity = O((V + E) log V)** using Min-Heap (Priority Queue)

This is the most optimized version using an **adjacency list** and a **binary heap**.

> ⚠️ Note: For **dense graphs** (where `E ≈ V²`), Dijkstra's becomes slower.

Coin change

📌 Greedy Coin Change – Pseudocode (Minimum Number of Coins)

plaintext

Copy Edit

```
function findMinCoinsGreedy(D[0...m-1], n):
    // D = array of coin denominations
    // m = number of denominations
    // n = target amount

    Sort D in descending order  // Use largest coins first
    S = empty list              // Stores the coins used

    for i from 0 to m - 1:
        while n ≥ D[i]:
            S.append(D[i])
            n = n - D[i]

        if n == 0:
            break

    if n ≠ 0:
        return "No solution"
    else:
        return S
```

✅ Time Complexity of Greedy Coin Change Algorithm

Let:

- `n` = target amount
- `m` = number of coin denominations
- `D[]` = array of coin denominations

1. **Sort denominations in descending order:**
   - We sort `m` coin values →
     - 🔹 **Time =** `O(m log m)`
2. **Iterate through the sorted coin array:**
   For each denomination `D[i]`, subtract as many times as it fits into `n`:

```plaintext
while n ≥ D[i]:
    n = n - D[i]
    S.append(D[i])
```

- This loop runs at most **O(n)** times in total (since each subtraction reduces `n`)
- Even though it's nested in a for loop, the total number of subtractions across all coins is **at most n/Dmin**, which is ≤ `n`

🔹 **Time = O(n)**

🔚 Total Time Complexity:

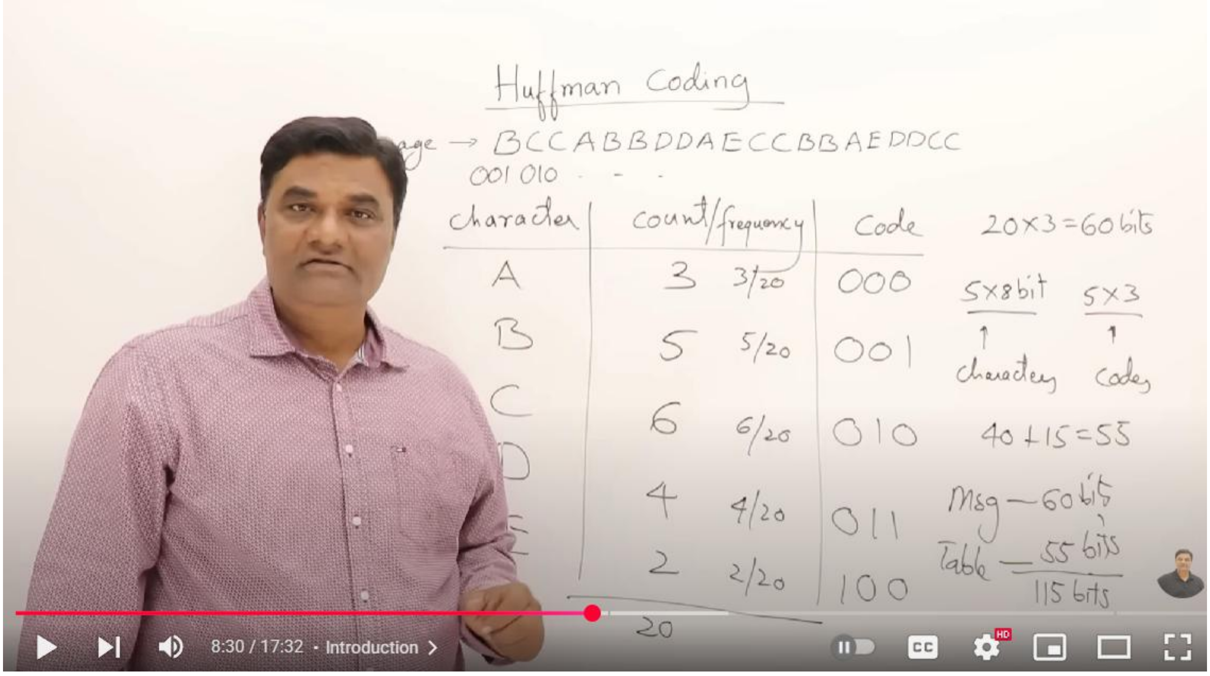
```plaintext
O(m log m + n)
```

✅ Final Answer:

> **Time Complexity = O(m log m + n)**

> - **"`m log m` for sorting"**
> - **"`n` for reducing the amount using the coin denominations"**

Huffman code



Youtube link :- https://youtu.be/co4_ahEDCho?feature=shared

📌 Huffman Coding – Pseudocode

plaintext

```
function huffmanCoding(charFreq[1...n]):
    // charFreq: array of (character, frequency) pairs

    Create a priority queue Q (min-heap) to store trees
    for each character c in charFreq:
        create a node with c and its frequency
        insert the node into Q
```

```
    while Q.size > 1:
        // Step 1: Extract two nodes with the smallest frequencies
        left = Q.extractMin()
        right = Q.extractMin()

        // Step 2: Create a new internal node with combined frequency
        newNode = new node
        newNode.frequency = left.frequency + right.frequency
        newNode.left = left
        newNode.right = right

        // Step 3: Add the new node back to the queue
        Q.insert(newNode)

    root = Q.extractMin()  // The root of the Huffman Tree
```

```
    // Step 4: Traverse the tree and assign codes
    codes = empty map
    assignCodes(root, "", codes)

    return codes


function assignCodes(node, currentCode, codes):
    if node is a leaf:
        codes[node.character] = currentCode
        return

    assignCodes(node.left, currentCode + "0", codes)
    assignCodes(node.right, currentCode + "1", codes)
```

✅ Time Complexity of Huffman Coding

Let:

- `n` = number of unique characters
- Each character has a frequency (char, freq)

Steps & Time Complexity:

🔸 **1. Build Min-Heap from `n` characters:**

- Insert all `n` nodes into the priority queue
  - 🔹 Time = `O(n)`

🔸 **2. Extract and merge nodes to build tree:**

- We perform `n - 1` merge operations
- Each `extractMin()` and `insert()` in heap = `O(log n)`
  - 🔹 Time = `O(n log n)`

🔸 **3. Assign binary codes by traversing the tree:**

- A single DFS traversal
  - 🔹 Time = `O(n)` (each node is visited once)

🔚 Total Time Complexity:

plaintext Copy Edit

```
O(n log n)
```

✅ Final Answer:

> **Time Complexity = O(n log n)**

Because the dominant step is the heap operations (extract + insert) during the tree-building phase.